```
        stream << "Triangle(" << shape.tr.d_side1 <<
                  ", "        << shape.tr.d_side2 <<
                  ", "        << shape.tr.d_side3 << ')';
      } break;
      default: {
        stream << "Error, unknown discriminator value: " << shape.tg.d_type;
      } break;
    }
    return stream;
}
```

An application using this framework would look quite similar to the program we saw previously for the `VShape` framework except that, instead of constructing a derived-class object, we construct one of the specific shapes and assign it to a `UShape` member before passing the `UShape` to a polymorphic subroutine:

```
void doSomethingU(const UShape& shape);  // arbitrary subroutine on a UShape

void testU()
{
    UShape u;                      // Default-initialize union; tg is active.
    u.ci = UCircle{ k_CI, 3.0 };  // Assign a concrete shape to a union member.
    doSomethingU(u);               // Invoke function on a circle via UShape.
    // ...
}
```

Importantly, all of the **`struct`**s participating in our vertically encoded **`union`** are of standard-layout type. Moreover, it so happens that all of the **`struct`**s are also of trivial type. Being both standard-layout and trivial, these **`struct`**s meet the definition of POD. What's more, because they comprise only public~~,~~ non**`static`**~~,~~ data members, they are syntactically and structurally compatible with **`struct`**s in the C language; with the addition of a few **`typedef`**s (e.g., **`typedef struct UCircle UCircle`**), the same declarations can be compiled by both C and C++ compilers to produce data structures whose source code is interoperable between the two languages. That said, the technique shown here can be modified slightly to work with standard-layout types that are *not* trivial, and, therefore, not POD types; see *Vertical encoding for non-trivial types (standard layout)* on page 448.

Note that the **`union`**-based `UShape` design has a somewhat different usage model than its protocol-based `VShape` counterpart. While the `VShape` base class does not depend on the set of concrete derived-class shapes, just the opposite is true for `UShape` and the set of concrete shape **`struct`**s. Hence, unlike with `VShape`, the `UShape` model doesn't offer reduced **physical dependencies** for clients that merely operate on shapes compared to those that create them.

Also note the different maintenance trade-offs: In the object-oriented design, adding a new function for all shapes affects every concrete shape derived from `VShape`, whereas in the **`union`**-based vertical-encoding design, adding a new shape affects every common operation on shapes and requires adding a new enumerator to the type tag. The primary advantages